CS 415 Operating Systems

Project 1 Report Collection

Submitted to:

Prof. Allen Maloney

Author:

Itay Mevorach

itaym

951918487

Report

Introduction

In this project we built a shell program. A shell program is a program which hosts the most simple and basic interface that an operating system can offer. There are many commands that a modern shell might have, though for this project we only implemented eight commands, as well as an exit feature. The commands implemented in this project allow for our basic interface interact with the operating system's storage spaces.

Modern shell functionalities include more advanced features. To name a few, "rename" is a function that conveniently renames files. A more interesting function, top shows current CPU processes, until an interrupt is caused. And finally, netstat is a useful command which displays a variety of information about your computer's network status. Building some of these commands is likely out of my current reach right now, however starting off with building a simple version of a shell program has deepened my understanding of operating systems. Additionally, this project helped me learn about the interaction of I/O with the operating system, through first-hand experience building a I/O shell program.

Background

The heart of this program is in the "command.c" file. This file was required to only use C system calls. To overcome this limitation, I decided to use the ***ssize_t write (int fildes, const void *buf, size_t nbyte)*** *system call to reflect the outputs of the command. In short, the write system call directly writes data from a specified buffer, const void *buf, to a designated file described by int files. Furthermore, the write system call requires a size_t nbyte argument, which describes the size of the buffer, const void *buf, as a size_t data type. I was considering using a macro for the write function since it was used too often in my command.c implementation. An example of how the write system call was used in the "command.c" file is shown in the Implementation section of this report.*

Source: https://pubs.opengroup.org/onlinepubs/009696699/functions/write.html

An essential C header used in "string_parser.c" is "string.h", which enables the use of commands such as printf and strtok_r. The ***char* strtok_r (char* _str, const char* _sep, char** _lasts)*** *function became an essential part of my implementation of "string_parser.c". The strtok_r function returns a token in a string char* _str, which we input as our buf argument, separating the sting with delimiter char* _sep, which we represent with our delim argument. In our usage of strtok_r, buf is the string we are looking for tokens in, separated by delim which, as mentioned prior, is our input for the char* _sep argument. Another feature of strtok_r is that it internally saves the pointer of where it last found a token in char* _str in char** _lasts, which we used a variable saveptr to store. My implementation and usage of strtok_r in the str_filler function is primarily from stack overflow as well as the source listed below.*

Source: https://pubs.opengroup.org/onlinepubs/009696699/functions/strtok_r.html

Implementation

The code below is my implementation of the "pwd" command in the "command.c" file. I chose this function out of the whole file as it features the most diverse uses of ***ssize_t write (int fildes, const void *buf, size_t nbyte)*** *and allows me to talk about all the features that write() has and its differences from printf(). The first and most obvious difference is that write() is a system call, while printf() comes from the "string.h" C header. Another key difference in write() is the functions first argument, in our case: STDOUT_FILENO. This argument essentially tells write() where to send the return of the function call. In our case, STDOUT_FILENO tells write() to send whatever we effectively print to stdout, or 1. In my implementation of "main.c" I change the stdout to "output.txt" when* ***./pseudo-shell -f*** *<> is called or, in other words, file mode is entered.*

```
void showCurrentDir() {
    char path[1035];
    if (getcwd(path, sizeof(path)) != NULL) {
        write(STDOUT_FILENO, path, strlen(path));
        write(STDOUT_FILENO, "\n", 1);
    } else {
        write(STDOUT_FILENO, "Error! Failed to run pwd command\n",
         strlen("Error! Failed to run pwd command\n"));
    }
}
```

*Some things to note about the varying usages of write() in this function are the different types of inputs used for the const void *buf argument. In the usage of write(), we use a variable called path, which is filled by the getcwd() system call, previously in the function. In the second usage of write(), we used a string input "\n" to create a new line for formatting, and our input for size_t nbyte was just the integer 1. This is because we already know that the const void *buf input will always be of length 1, and thus can hard-code it into the function. In the last usage of write(), we inputted a constant string with an error message, and instead of counting how long the message was I opted to simply use strlen() and copy the message into the function.*

Performance Results and Discussion

My program has accurate outputs for every command implemented. At first my program had a segmentation fault when getting to the "cp" command in the input file. I found out through some debugging that I had forgot to implement a case for when cp was looking to copy a file into a directory. I ended up implementing an if statement that uses the chdir() system call to check if the destinationPath argument was a directory, by comparing to -1, the function fail return value of the chdir() system call. This allowed my "cp" command to have a different scenario for copying into a directory.

Conclusion

In conclusion, this project was rigorous and showed just how complicated even the most basic program of an operating system really is. I personally learned a lot about system calls and parsing through tokens. This has definitely been a tasking project and proved to be a worthy learning experience.